Fine-Grained Version Control in COOP/Orm

Boris Magnusson

LU-CS-TR:96-163

Also presented at: European Conference on Computer Supported Cooperative Work 1995 in Workshop on Version Control in CSCW Applications, Stockholm September 1995

Department of Computer Science
Lund Institute of Technology
Lund University

P.O. Box 118, S-221 00 Lund
Sweden

Fine-Grained Version Control in COOP/Orm.

Boris Magnusson
Dept of Computer Science, Lund University
Box 1 18, S-221 00 Lund, Sweden
e-mail: Boris @dna.lth.se

Position paper . We are discussing the role of version control in collaborative applications. W e argue that many of the mechanisms needed in a collaborative editing environment can be built on top of a fine-grained version control mechanism and provide the integration mechanism. W e outline how such a system has been designed in the COOP/ Orm effort.

1 Background

When sharing information with other people, doing simultaneous changes, versions of the shared information becomes a very critical issue. Not only do we have to answer questions like *What changes did I do to the program (since it is no longer working)?* , or *What changes have I done since the last release?* , but also *What have happened to the system during my vacation?* and *What changes did my colleague do to the paper since I saw it last time?* Such questions regarding the history of a document are very frequent, one should expect them to be easy to answer and that such facilities are considered basic functionality . In common environment, where people actually work together to develop technical systems such as software systems, structured documentation etc. this is not so. W e will here first try to summarize the needs related to version control in collaborative environments and then describe how we in COOP/Orm have set out to try to solve these problems.

2 Problems and demands

Integrated representation.
Version control systems are traditionally implemented on top of an existing file system, as a secondary concern. The primary representation of a document is a snapshot of its development and several versions are realized through several files. The tools in the environment (editors, compilers, text processors etc.) do not understand versioning. This way of representing versions of documents as independent files thus limits the support that can be offered to the user . We have chosen a model where the representation of a document include its history with all its versions. As a consequence all tools in our environment must understand this representation and version control.

Support for structured documents
Most interested documents are structured in the sense that they are made up of small information units that a related to each other . A very common structure is

hierarchies, or trees. Book with chapters, sections and paragraphs with text or figures is one example, programs with classes and procedures is another important example. Typical for such hierarchical documents are that they share revision history in the sense that a version of composite (e.g. second printing of a book) determines the version of all its components and that a change to one of its components creates a new version of the composite. Traditional version control systems have no notion of hierarchical documents and the user have to make the choice between creating large documents, which are versioned controlled as one unit, but can be awkward to work with or share, or to create many small components and to manage the versioning of the composite manually. Some recent version control systems such as CVS have acknowledged the need for collection of documents with common version history and offers versions of Unix directories as a mechanism for this. Again we have chosen to integrate support for structured documents into the fundamental storage model.

Versioned configurations

Most systems are actually configurations of smaller parts. One important reason for this is to make it possible to include a component into several different systems. Program libraries is a trivial example of this situation as well as reuse of drawings and illustrations in papers and articles. Configurations are different from structured documents since a change to one component should not automatically generate new versions of all systems where it is included. If it does, which is often the case in simple check-in/check-out versioning systems, the possibility to track the history of the configurations is lost. The mechanism needed here is to have links which include information about the version of the destination document. A pre-requisite is that all versions of a document are readily available. Again, these mechanisms are included in our model.

Alternatives and merge

In our opinion, pessimistic check-out ('locking') is too restricted to use in a collaborative, distributed environment. The alternative is to allow parallel development of alteratives and then provide easy to use facilities for merging. Alternatives and merging also appears from planned development and needs to be supported anyway, but allowing optimistic creation of new alternatives will make these, as well as merging, more frequent. Current version control systems focus on manageing changes to information in single files. Changes to the structure of a large document, or system, is not supported and can be very akward to perform in a multi-user environment.

Intuitive user interface

We have argued that facilities for seeing old versions, comparing versions, and indeed also merging documents are fundamental and frequently needed already in a single user environment and much more so in a collaborative environment. It should thus be simple and easy to perform these tasks. Our model of active diffs, interactively presenting differences between documents, is providing a uniform and intuitive model. As we shall see below it is used during editing, when comparing versions, during merge and in synchronous editing mode. A graphical presentation of the version graph makes an overview of the development history of the document and to navigate and compare versions, also widely apart.

Distributed shared documents
Users may be geographically distributed and need to work also in case of network failure, although there might be some degradation of system capabilities. In order to support such a situation, the storage model must cope with replication and later merge of documents. Our model including the full history of a document as an identifiable unit, rather than each individual version, makes it almost trivial to synchronize modified replicas because each modification is always just an addition of versions.

Modes of synchronization
Teams of users tend to work with different need for synchronization during different phases of a project. There is thus a need for support for a variety of synchronization modes, from asynchronous (where each user work independently) to synchronous where users can see what others are doing with a fine grain of detail, both regarding time and detail of modifications. It should be possible to without much effort to change between different modes of interaction.

User awareness
Users in collaborative system needs to be aware of what other users are doing, or has been doing in the system. The level of detail needed depends on the mode of synchronization the user is engaged in. In our model modifications are always represented through creation of a new version of the document. All users share the same version graph which give the lowest level of user awareness. In synchronized mode each single modification can be shown through active diffs.

3 The COOP/Orm model for cooperative applications

An analysis as the one presented in the previous section have resulted in a model built on a fine-grained version control mechanism. We will here very briefly describe the model and discuss how it answers the needs outlined above.

The COOP/Orm model is built on the idea of representing the complete development history of a document. Rather than having some versions of a document represented in full and other versions as deltas all this information is stored in one document. The storage model is also supporting a hierarchical structure of information. Retrieving information – the client has to provide the tree address of a node in the hierarchy and a version identifier. The result is a representation of the information in the node in full, and if it not the latest revision of the node in an alternative is requested, also some deltas. With this information the client can recreate the version of the node it asked for. Storing information the client has to provide both the node in full version, and backwards delta to the version it started from.

The storage representation is compact since it is only storing the full version of an information node in the last revision of a node in an alternative. Older versions of the node are represented by backwards deltas. Nodes that are unchanged between versions are shared. Unchanged nodes are also shared between alternatives, which makes creation of an alternative a cheap operation (while in other models it means copying full documents). It should be noted that in this model a version of a document can never be changed, the only way to change a document is to create a new version.

Structural representation

The storage model support managing the version history and the hierarchical structure but the information stored is not fixed in the model. This means that it can be used by a variety of editors for text, graphics, syntax trees etc. An editor provides information nodes in full and delta form. The same editor is responsible to combine information and deltas to recreate a version of an information node.

The storage model include information about which versions exist and how they relate to each other . It therefor support construction of a version graph. Comparing versions on the structural level it is immediately evident from the model which nodes are unchanged between the versions since they are shared. For each node in a document it is also possible to request a ’local version graph’. This is a presentation of in which versions of the document this particular node is actually changed. The version graph, which is dynamically updated, is the primary source of user awareness in our system.

Providing redundance in a distributed environment is simple through replication. Each replica might, during a network failure, evolve through addition of new versions. Later merging is almost trivial since each replica only have additions of new versions, starting from some common state.

Editors

The editors in our system are responsible for presenting, editing and storing the information in a particular node and is thus free to determine its storage format. The current text editor we are using is supporting the metaphor of ’active diffs’ to present differences between two versions of a node. This is used to mark the chnages made in the current editing session (comparing with the start version), to present differences between any two sequential versions, and to present differences during merge of two alternatives. Merging is guided by a set of default rules to calculate a suggested merged version and to identify conflicts. Two alternatives can be ’hypothetically merged’ to explore the potential conflicts. This mechanism together with a mechanism to communicate changes from one user to another gives a very detailed level of user awareness. It is used in synchronous mode of collaboration to present what other users are doing. It should be noted that each user is free to do whatever changes he want since he is creating his own version. Although we have so far only worked with an editor for text, the model support arbitrary information. We plan to add editors for abstract syntax trees and graphics.

Editing on the structural level, e.i. adding and removing information nodes and ’folders’ (nodes that only contain other nodes, e.g. representing chapters). Changes and differences here are represented using the same ’active diff’ approach as used by the contents editors. Diffs on this level makes it easy to identify changes also in tree structured environment, in particular to recognize sub-trees that are equal and thus of less interest.This mechanism provides an intermediate level of user awareness, between the version graph and the detailed information in a node.

4 Development and future work

The basic mechanisms of COOP/Orm has been designed and implemented and are now operational. The interaction model supporting both asynchronous and semi-synchronous collaboration is described in more detail in a paper at last ECSCW [MM93]. The storage model has been described from a technical perspective in [MAM93] and in [Ask94]. Finally the text editor has been presented in [Ols94]. The implementation work has now converged to a client-server architecture and the interactive, update of active diffs in the synchronous collaboration mode is the next step. With this step completed the COOP/Orm collaborative editor can be used for evaluating the functionality in a multi user setting. Integration with traditional tools and environments, which are demanding files with single version documents, can be achieved through dumping complete versions to file. The COOP/Orm system can thus be used as a front-end in existing environments.

The server-server protocol has been designed and outlined in [MG95], but not yet implemented. The design is focusing on distributed aspects and on collaborative aspects on the configuration management level.

Future development includes re-hosting our grammar interpreting syntax directed editor (SbyS) to this fine-grained version control environment. This will allow us to have control of versioning of both the document and the defining grammar, and thus unique control of consistency relations in evolving formal notations. We are thinking not only of traditional programming languages, but also 'application languages' which are much more likely to change rapidly.

An other aspect, perhaps more central to CSCW, is support for the work process. Here we plan to explore the possibilities of using formal descriptions defined by our syntax directed environment, but achieve a unique level of flexibility and tailorability by allowing the users to modify, extend, these descriptions. Using the version controlled grammar descriptions will, hopefully, make it possible to maintain some level of consistency in such a situation and at the same time give the users enough freedom not to fell restrained. In such an environment one could envision that the users in the end define the work process, rather than the managers, who too often know to little of the work involved. The consequence would be that the work process descriptions that emerge could be seen as synthesis of experience rather than prescriptions in advance.

References

[Ask94] Ulf Asklund. Identifying Conflicts During Structural Merge. In Magnusson et al. MHM94.

[MAM93] Boris Magnusson, Ulf Asklund, and Sten Minör. Fine-Grained Revision Control for Collaborative Software Development. In *Proceedings of ACM SIGSOFT'93 - Symposium on the Foundations of Software Engineering*, Los Angeles, California, 7-10 December 1993.

[MA95] Boris Magnusson and Ulf Asklund. XXX: A Model for Fine Grained Version Control of Configurations. Draft report, Lund University 1995.

[MG95] Boris Magnusson and Rachid Guerraoui. Support for Collaborative

Object-Oriented Development. Draft report, Lund University 1995.

[MHM94] Magnusson, Minör and Hedin (Eds.): Proceedings of Nordic Workshop on Programming Environments, Lund 1994, LU-CS-TR:94-127.

[MM93] Sten Minör and Boris Magnusson. A Model for Semi-(a)Synchronous Collaborative Editing. In *Proceedings of the Third European Conference on Computer Supported Cooperative Work*, Milano, Italy, 1993. Kluwer Academic Publishers.

[Ols94] Torsten Olsson. Group Awareness Using Fine-Grained Revision Control. In Magnusson et al. MHM94.